CPU/ATI GPU 混合体系结构上 DGEMM 的性能研究

李佳佳 李兴建 谭光明

摘要： 本文报道了我们在 CPU/ATI GPU 混合体系结构上优化双精度矩阵乘法（DGEMM）的工作。在真实应用中， CPU 与图形处理器（GPU）之间的数据传输是影响性能的关键因素。由于软件流水可以降低数据传输开销，我们提出了三种软件流水算法，分别是双缓存（Double Buffering）、数据重用（Data Reuse）和数据存储优化（Data Placement）。在 AMD 公司的图形处理器（GPU）ATI HD5970 上，优化后 DGEMM 性能达到 758 GFLOP/s，对应效率为 82%，是 ACML-GPU v1.1 性能的两倍。在 Intel Westmere EP 和 ATI HD5970 组成的异构系统上，性能达到 844 GFLOP/s，效率为 80%。我们进一步考察了多个 CPU 和多个 GPU 上 DGEMM 的扩展性，详细分析了体系结构方面的影响因素。分析表明，PCIe 总线和内存总线的竞争是异构系统上程序性能降低的重要影响因素。

关键词 高性能计算 GPU CAL 矩阵乘法

1 引言

双精度矩阵乘法是影响科学和工程领域多种应用性能的重要因素。多种关键数值算法，如 BLAS3[1][1]和 LU 分解[2]，都依赖于 DGEMM 的高性能实现。而全球超级计算机排名的依据——HPL[3]测试程序则是由稠密矩阵的 LU 分解构成。由于 DGEMM 的性能很大程度上依赖于计算机硬件，许多处理器厂商开发了基于特定机器的 DGEMM，如英特尔的 MKL 和 AMD[2]的 ACML，并且在各自的多核处理器上进一步优化 BLAS 库。图形处理器（GPU）的浮点峰值性能比通用 CPU 高出一个数量级以上，例如 AMD HD5970 双精度浮点性能达到 928 GFLOP/s；NVIDIA Tesla C2070 的双精度浮点峰值性能为 515 GFLOP/s。众所周知，DGEMM 是计算密集型程序，访存比较规律，这使其适合在 GPU 上进行优化。事实上，已经有许多在 GPU 上优化 DGEMM 的工作[11-17,19-21,23-27]。以前的工作大多数针对 DGEMM 的计算矩阵已经存储在显存上的情况，对 DGEMM 在 GPU 上的实现进行优化。当前，在许多系统中 GPU 通过 PCIe 总线作为加速部件与 CPU 相连。在实际应用中，初始时 DGEMM 的计算矩阵都存放在 CPU 主存上。由于 GPU 只能使用存储在 GPU 显存上的数据，因此 GPU 计算 DGEMM 之前需要进行 CPU 与 GPU 之间的数据传输。由于 GPU 显存容量的限制，大规模数据需要 CPU 与 GPU 之间进行多次数据传输。

CPU/GPU 的异构系统中，存储层次之间的数据传输对 DGEMM 的性能有重要影响。从系统的角度看存储层次分为两层：CPU 的主存和 GPU 的显存。首先，CPU 通过 PCIe 总线向 GPU 显存传输数据，然后 GPU 渲染器（shader）从显存取数据计算。GPU HD5970 的显存峰值带宽为 256 GB/s，而 PCIe 总线的峰值带宽是 8 GB/s。虽然 DGEMM 是计算密集型，但这两个带宽的差异仍然是异构系统上 DGEMM 总体性能的瓶颈。中里直人（N. Nakasato）在引文[15]中指出，如果将 CPU 与 ATI Cypress GPU 之间数据传输时间计入 DGEMM 的总时间，DGEMM 的效率会从 85%降至 55%。AMD 基于 CPU-ATI GPU 异构系统开发的 GEMM

[1] Basic Linear Algebra Subprograms 基础线性代数程序集
[2] 超威（港台通常译作“超微”）

优化库，也存在这种效率下降现象。

优化 DGEMM 和分析未来混合体系结构的趋势，都需要对每个存储层次进行定量分析：了解当前的 CPU-GPU 异构体系结构，存储层次对 DGEMM 的总体性能到底产生了多大影响。虽然之前已经有一些 DGEMM 在 GPU 的优化工作，但少有文章进行了定量的分析，尤其对于 CPU/GPU 异构系统。

本文通过对多核 CPU/Cypress GPU 异构系统的研究来解决上述问题。通过设计新算法来减少存储层次间数据传输的开销，我们在这个异构系统上得到了 DGEMM 的快速实现。本文的主要贡献有三点：

- 定量分析了异构体系结构上的 DGEMM 性能，其中详细分析了 CPU-GPU 之间数据传输过程。分析显示，数据传输过程占 DGEMM 总时间的 40%，而不是先前工作得到的 20%，因此数据传输的开销对 DGEMM 的性能有很大影响。
- 针对大规模的 DGEMM 提出了新的流水算法。三种优化方法依次是：双缓存优化（Double Buffering）、数据重用优化（Data Reuse）、数据存储优化（Data Placement）。优化后的 DGEMM 在一个 Cypress GPU 上，性能达到 408GFLOP/s，对应效率达到 88%。优化后的 DGEMM 性能是 ACML-GPU v1.1 的 2 倍多。在 ATI HD5970 上，优化后的 DGEMM 性能达到 758GFLOP/s，对应效率为 82%；混合版 DGEMM 在 Intel Westmere EP/ATI HD5970 的异构系统上，性能达到 844GFLOP/s，为峰值性能的 80%。
- 最后，我们分析了将 DGEMM 扩展到多个 CPU 和多个 GPU 时的资源竞争情况，重点分析了 PCIe 竞争和内存竞争。从中得出，资源竞争（PCIe 竞争和内存竞争）是可扩展性的主要瓶颈之一。然而，单纯凭借软件方式不能完全避免资源竞争对 DGEMM 整体程序的影响。

2 背景

我们的性能优化主要针对 ATI Evergreen GPU 体系结构，Cypress 是 Evergreen 系列中最前端的 GPU。ATI HD5870 卡封装了一个 Cypress 卡，而 HD5970 集成了两个 Cypress 卡。这部分主要介绍 Cypress 的几个重要特性——ATI CAL[3][6]软件层面上的存储层次。我们将 ACML-GPU 库中的 DGEMM 程序作为基准程序，并以它为基础进行性能优化，因此我们对 ACML-GPU 中 DGEMM 进行详细分析，从中寻找优化途径。

2.1 Cypress GPU

一个 Evergreen GPU 芯片集成了多个计算单元、一个控制单元（又称为线程分发部件）、存储控制器和 DMA[4]引擎。为了充分发挥浮点操作的吞吐量，Cypress GPU 微体系结构采用单指令流多数据流（SIMD）和超长指令字（VLIW）。由于

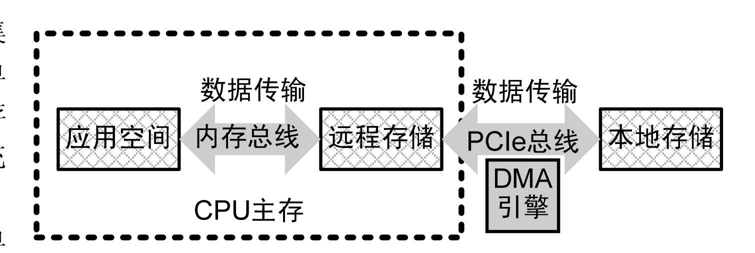


图1. CAL 系统在 CPU 与 ATI GPU 间的存储层次

[3] Compute Abstraction Layer，计算抽象层（亦有译作“计算提取层”）
[4] Direct Memory Access，直接内存访问

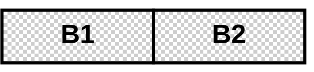
DGEMM 在 GPU 上的核心程序（kernel）[5]的优化不是本文的着重点，在这里我们不详细描述 GPU 微体系结构的细节。本文使用文献[15]中的核心程序，其效率为 80%。简单讲，一个 Cypress 卡当频率为 725MHz 时单精度浮点峰值性能为 2.32TFLOP/s，双精度浮点操作性能是单精度性能的 1/5，因此在 725MHz 频率下双精度浮点操作的峰值性能为 464GFLOP/s。

图 1 画出了 CAL 软件系统层面 CPU/ATI GPU 的异构体系结构的存储层次。下面我们详细介绍使用 CAL 系统中与程序优化密切相关的存储层次特征：

- CAL 系统将物理存储划分为本地存储和远程存储两部分。本地存储指显存，为显卡上的高速存储。远程存储指物理存储空间不在显卡上，但 GPU 仍可访问的存储空间（即 CPU 主存的部分空间）。本地存储和远程存储都可由 GPU 核心程序读取。换言之，GPU 核心程序既可以把 GPU 寄存器中的数据写回显存，也可直接写回远程存储。由于对远程存储空间的操作相对本地存储的操作有更长的延迟，因此这种存储方式的性能较低。远程存储进一步分为可缓存（cached）部分和不可缓存（uncached）部分。例如，CAL 系统将 HD5970 的远程存储划分为 1788MB 的不可缓存存储和 500MB 的可缓存存储。因此，**选择哪种存储空间作为 CPU 与 GPU 之间的共享数据空间，将会影响程序的性能。**
- CAL 应用中初始数据存放在 CPU 的应用空间。GPU 核心程序使用数据前需要进行两次数据传输：应用空间和远程存储之间与远程存储和本地存储之间。一种优化方法是利用系统的固定内存（pinned memory）——应用程序直接将 CPU 应用空间的数据传输到远程存储空间，使其直接进行 DMA 传输。这种方法省去了从应用空间到远程存储的数据传输。一些应用使用 CUDA[6] [5]实现，通过这种方法有效地提升了性能。在 CAL 系统中，同样存在固定内存的技术，但固定存储在大小等方面存在限制。因此，**通过存储层次之间适当的组织（如流水线算法）来降低 PCIe 总线上的数据传输时间是十分必要的。**

2.2 DGEMM

本节描述大规模 DGEMM 在 CPU/ATI GPU 异构系统上的算法，原始数据存储在 CPU 的应用空间。DGEMM 计算 $\boldsymbol{C} := \boldsymbol{alpha} \times \boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{beta} \times \boldsymbol{C}$，其中 $\boldsymbol{A}$、$\boldsymbol{B}$ 和 $\boldsymbol{C}$ 分别是规模为 $m \times k$、$k \times n$、$m \times n$ 的矩阵。涉及 DGEMM 的实际应用中，这三个矩阵的规模较大，不能全部存储在 GPU 显存上。因此，这三个矩阵被划分为多个子矩阵块，每次将几个矩阵块传输到GPU 显存，进行部分 DGEMM 计算。划分算法将 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$ 划分为 $\boldsymbol{A} = \{\boldsymbol{A}_1, \boldsymbol{A}_2, \cdots, \boldsymbol{A}_p\}$，$\boldsymbol{B} = \{\boldsymbol{B}_1, \boldsymbol{B}_2, \cdots, \boldsymbol{B}_q\}$，$\boldsymbol{C} = \{\boldsymbol{C}_1, \boldsymbol{C}_2, \cdots, \boldsymbol{C}_{p \times q}\}$，其中 p 和 q 依赖于显存大小。以 p=2，q=2 为例，矩阵块乘法如图 2 所示。



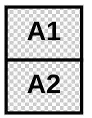


图2. 矩阵块划分

图 2 中划分产生了四个独立的 $\boldsymbol{C}$ 子矩阵，$\boldsymbol{C}_1 = \boldsymbol{A}_1 \times \boldsymbol{B}_1$、$\boldsymbol{C}_2 = \boldsymbol{A}_1 \times \boldsymbol{B}_2$、$\boldsymbol{C}_3 = \boldsymbol{A}_2 \times \boldsymbol{B}_1$、$\boldsymbol{C}_4 = \boldsymbol{A}_2 \times \boldsymbol{B}_2$，这四个矩阵块运算相互独立，可以并行计算。对每个子矩阵乘法，我们将其依赖的 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 子矩阵传入显存，DGEMM 核心程序将这些子矩阵进一步划分为更小的矩阵 [9-13]（如缓存（cache）分块和寄存器分块）。基于 ATI CAL 系统的存储层次，大规模 DGEMM 实现如算法 1。远程存储作为 CPU 和 GPU 之间的共享空间，CPU 的应用空间与 GPU 的显存（本地存储）之间传输数据必须经过远程存储（第 1 行）。算法 1 的伪代码中，加载数据到 GPU 包

[5] kernel：在 GPU 上实现的 DGEMM 核心程序
[6] Compute Unified Device Architecture，一种由英伟达（NVIDIA）推出的通用并行计算架构

含两个步骤（*load*$_1$ 和 *load*$_2$），将数据写回 CPU 内存为 1 个步骤（*store*）。事实上，第 5 行 DGEMM *mult* 核心程序的操作中隐式地包含了另一个数据写回操作——子矩阵 C 的数据从 GPU 寄存器写回到远程存储。

```
Partition: A={A1, A2, …,Ap}, B={B1,B2,…,Bq}, C={C1, C2, …,Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2,…}
////////////////////////////////////////////////////////////////////////
1. bind remote memory for sub-matrices A,B,C
2. for each workunit wui do //i=1,2,…,pxq
  //load1
3. copy both Ai and Bi from application space into remote memory
  //load2
4. copy both Ai and Bi from remote memory to local memory
  //mult
5. calculate Ci on GPU device and directly output it to remote memory
  //store
6. copy Ci from remote memory to application space (also multiply by beta)
7. endfor
```

算法1. DGEMM 的原始实现

我们将算法 1 作为进行性能对比的基准程序。下面我们给出异构 CPU/GPU 系统上 DGEMM 运行时各部分的详细剖析。图 3 给出了算法 1 的每部分时间。由于不同问题规模下 DGEMM 各部分时间比例有着近似的分布，我们以 ***k***=2048 为例，x 轴值为 ***m***(***n***)表示矩阵规模。我们发现 GPU *mult* 核心程序占用了大部分时间（大于 70%），而其余的三部分数据传输总共占据不到 30%的运行时间，直观上数据传输的时间可以被核心程序的计算时间所掩盖。为了寻找可并行执行的操作，我们将 DGEMM 算法中使用资源分为：GPU，CPU+内存总线，PCIe 总线三部分，这三种资源间的操作可以并行执行。

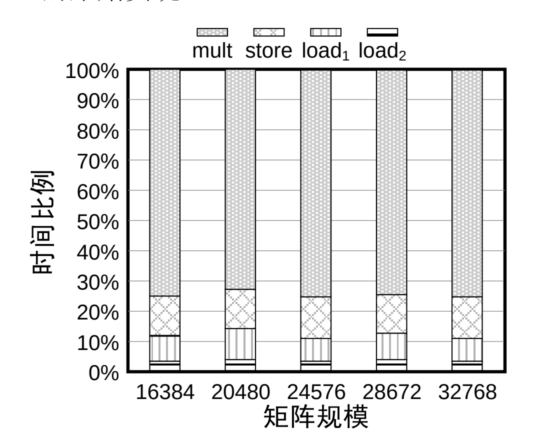


图3. DGEMM 原始实现的时间组成

表 1 给出了算法 1 中各部分的资源占用情况。*load*$_1$ 和 *store* 占用 CPU+主存总线，*load*$_2$ 只需要 PCIe 总线来传输数据。*Mult* 核心程序在 GPU 上执行，通过 PCIe 总线将结果输出到 CPU 主存。根据资源分配情况，利用软件流水算法来重叠数据传输（*load*$_1$，*load*$_2$，*store*）和 *mult* 核心程序是可行的。杨灿群（Canqun Yang）等人也实现了利用流水线算法使 *load*$_1$ 和 *mult* 重叠执行[21]。然而，正如他们文章中所说及本文第 4 部分的实验结果显示，简单的流水线算法对性能的提升并不明显（大概 20%）。事实上，算法 1 的执行时间主体为 *mult* 核心程序。先前算法中 *mult* 核心程序都是直接将计算结果从寄存器写回远程存储，其中 *mult* 核心程序部分除了包含矩阵乘法计算还包含从 PCIe 总线写回数据。这样，在使用流水线后，核心程序中的数据传输成为性能瓶颈。因为：（1）. PCIe 带宽小于 GPU 显存/CPU 主存带宽；（2）. *mult* 写回的数据会比 *load*$_2$ 操作中传输的数据大。例如 LINPACK 中的 DGEMM，***k*** 远小于 ***m*** 和 ***n***，因此输出的 ***C*** 矩阵大小 ***m***×***n*** 大于输入的

表1. 算法 1 中各步资源分配情况

	CPU+内存总线	GPU	PCIe 总线
$Load_1$	(灰)		
$Load_2$			(灰)
Mult		(灰)	(灰)
Store	(灰)		

矩阵 ***A***, ***B*** 的大小 ***k***× (***m***+***n***)。并且如下一节所说，可以通过进一步开发数据重用优化来降低 `load`$_2$ 中的数据传输次数。因此我们优化流水线算法使浮点操作和数据传输操作重叠得更好。

3 流水线算法

软件流水是使计算和存储操作重叠的通用方法。算法 1 中有 `load`$_1$、`load`$_2$、`store` 三个显式的存储操作在 CPU 和 GPU 之间传输数据，乘法操作（`mult`）直接输出结果到远程存储。因此，用流水线算法实现 `mult` 和三个存储操作的并行执行是必要的。

3.1 双缓存优化（Double Buffering）

```
Partition: A={A1, A2, …,Ap}, B={B1,B2,…,Bq},
C={C1, C2, …,Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2,…}
Cij: the sub-matrices C is partitioned into blocks
//////////////////////////////////////////////
1. bind remote memory for sub-matrices A,B,C
2. for each workunit wui do //i=1,2,…,pxq
  //load1
3. copy both Ai and Bi from application space
into remote memory
  //load2
4. copy both Ai and Bi from remote memory to
local memory
  //mult
5. calculate Ci,1 on GPU device and output it to
remote memory
6. for each block Ci,j do //j=2,3…
        //store
7.   copy Ci,j-1 form remote memory to
application space (also multiply by beta)
     //mult
8.   calculate Ci,j on GPU device and output it
to remote memory
9. endfor
     //store
10. copy the last Ci,j form remote memory to
application space (also multiply by beta)
11. endfor
```

算法2. 结合双缓存优化（Double Buffering）的 DGEMM

```
Partition: A={A1, A2, …,Ap}, B={B1,B2,…,Bq},
C={C1, C2, …,Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2,…}
Cij: the sub-matrices C is partitioned into blocks
//////////////////////////////////////////
1. bind remote memory for sub-matrices A,B,C
//pre-processing
Allocate workunits in a wriggled way
//the for-loop is pipelined
2. for each workunit wui do //i=1,2,…,pxq
  //load1
3. copy either Ai or Bi from application space
   into remote memory according to the
   indicators
  //load2
4. copy either Ai or Bi from remote memory to
   local memory according to the indicators
  //mult
5. calculate Ci,1 on GPU device and output it to
remote memory
6. for each block Ci,j do //j=2,3…
        //store
7.   copy Ci,j-1 form remote memory to
application space (also multiply by beta)
     //mult
8.   calculate Ci,j on GPU device and output it
to remote memory
9. endfor
     //store
10. copy the last Ci,j form remote memory to
application space (also multiply by beta)
11. endfor
```

算法3. 结合数据重用优化（Data Reuse）的 DGEMM

写回 ***C*** 矩阵的 `store` 操作实现了数据传输和少量的浮点计算(***beta*** × ***C***)，占用一定 CPU 资源。为隐藏写回 ***C*** 矩阵的开销，一种方法是在不同工作单元间实现流水。例如，当工作单元 i 执行 `store` 操作时，工作单元 i+1 的 `mult` 操作可以同时进行。这种流水线存在两个问题。首先，`load`$_1$ 和 `store` 之间存在资源冲突，因为他们都是在应用空间和 CAL 的远程

存储（CPU 主存的一部分）之间进行数据传输，资源的冲突降低流水线效率。其次，远程存储空间的大小有限，尤其是可缓存远程存储。由于 store 操作在 CPU 部分执行，因此利用可缓存远程存储空间保存产生的 ***C*** 子矩阵。但缓存空间大小限制流水线中可并发执行的工作单元数量。

一个更好的策略是在一个工作单元中开发更细粒度的流水线。算法 2 给出了细粒度流水线的伪代码，将生成的 ***C*** 子矩阵块进一步划分成更细的子块，这些子块用流水方式执行。我们开发了双缓存算法，在可缓存远程存储上开辟两个缓存空间。对于算法 2 第 6-9 行的每个 for 循环，store 操作将一个缓存的子矩阵块 $\boldsymbol{C}_{i,j}$ 写回应用空间，同时 mult 计算下一个子矩阵块 $\boldsymbol{C}_{i,j+1}$ 并将其写回到另一个缓存空间。流水过程中，mult 和 store 交替使用这两个缓存空间。由于 GPU 的核心程序以异步的方式执行，因此对于每个循环 mult 和 store 都可以并行执行。

3.2 数据重用优化（Data Reuse）

从上面得知，$load_1$、$load_2$ 和 store 操作的执行时间总和比 mult 执行时间小很多（图 3），这三部分的开销可以较容易地被工作单元间的流水线执行方式掩盖。然而，各操作间仍存在资源冲突，$load_1$ 和 store 之间存在 CPU 主存冲突，$load_2$ 和 mult 之间存在 PCIe 冲突。资源冲突导致流水线的延迟，从而降低 DGEMM 的整体性能。

幸运的是，在两个连续的工作单元间我们可以开发数据重用。以图 2 为例，如果我们按照 $\boldsymbol{WU}_1=\boldsymbol{C}_1=\boldsymbol{A}_1\times\boldsymbol{B}_1$， $\boldsymbol{WU}_2=\boldsymbol{C}_2=\boldsymbol{A}_1\times\boldsymbol{B}_2$， $\boldsymbol{WU}_3=\boldsymbol{C}_4=\boldsymbol{A}_2\times\boldsymbol{B}_2$， $\boldsymbol{WU}_4=\boldsymbol{C}_3=\boldsymbol{A}_2\times\boldsymbol{B}_1$ 的顺序调度工作单元，每两个连续的工作单元间都有一个输入子矩阵是相同的，这就降低了资源冲突的开销。开发数据重用还需要另外两个步骤：首先，需要一个预处理过程，对工作单元重新排序，结果存放在一个队列中。我们把 ***C*** 矩阵划分为条形矩阵的集合，以整数(i=0,1…)命名。每个条形矩阵再分为矩阵块，当 i%2=0 时自底向上划分；当 i%2=1 时自上而下划分。我们以这种“迂回”的方式划分条形矩阵，一个条形矩阵顶部（或底部）的矩阵块与下一个条形矩阵顶部（或底部）的矩阵块相连，并将所有矩阵块按序存入队列。其次，我们需要设置两个标志，标出正在运行的 ***A***、***B*** 的矩阵块在队列中的位置，避免载入相同的矩阵块。算法 3 给出了结合数据重用优化的流水线算法。

3.3 数据存储优化（Data Placement）

数据重用优化中 ***A*** 和 ***B*** 的矩阵子块临时存储在 GPU 本地存储中。我们将 ***A*** 和 ***B*** 的远程存储空间设置为不可缓存方式，因为:（1）. 矩阵 ***A*** 和 ***B*** 的数据传输过程中不需要计算；（2）. 远程存储的可缓存部分空间太小，不能有效加速 $load_1$ 操作的执行。由于所有工作单元中的 ***C*** 子矩阵之间不存在数据重用，是相互独立的，且 ***C*** 矩阵作为最终输出结果不会重新使用，这样看来，将 ***C*** 矩阵存储在远程存储而非 GPU 本地存储很合理，并且减少了有限显存空间的占用。因此上面的三个算法中，***C*** 矩阵都是存储在可缓存远程存储上，以期在诸如内存间数据拷贝和与 ***beta*** 的乘积等运算获得更高的 CPU 性能。

表2. 优化后 DGEMM（算法 4）的资源分配情况

	CPU+内存总线	GPU	PCIe 总线
$load_1$			
$load_2$			
mult			
$store_1$			
$store_2$			

上述流水线的执行使得数据传输操作（$load_1$、$load_2$、store）与乘法核心程序（mult）重叠，流水线的执行时间基本由 mult 时间决定。因此，目前的优化方向是降低流水线中

mult 的执行时间。由于 *mult* 核心程序中包含数据传输操作，输出数据从 GPU 寄存器直接写回远程存储。我们的优化策略是在流水线中增加一个额外的阶段，这样输出数据到远程存储的操作也可以和 GPU 核心程序的计算重叠。

Partition: $A=\{A_1, A_2, \ldots, A_p\}$, $B=\{B_1, B_2, \ldots, B_q\}$, $C=\{C_1, C_2, \ldots, C_{pxq}\}$
Work unit: $WU=\{C_1=A_1*B_1, C_2=A_1*B_2, \ldots\}$
C_{ij}: *the sub-matrices C is partitioned into blocks*
//
1. bind remote memory for sub-matrices A,B,C
//pre-processing
Allocate workunits in a wriggled way
//the for-loop is pipelined
2. ***for*** each workunit wu_i **do** *//i=1,2,…,pxq*
//load1
3. copy either A_i or B_i from application space into remote memory according to the indicators
//load2
4. copy either A_i or B_i from remote memory to local memory according to the indicators
//mult
5. ***DMAPipeline***($C_{i,1}$)
6. ***for*** each block $C_{i,j}$ **do** //j=2,3…
//store2
7. copy $C_{i,j-1}$ form remote memory to application space (also multiply by *beta*)
//mult
8. ***DMAPipeline***($C_{i,j}$)
9. ***endfor***
//store2
10. copy the last $C_{i,j}$ form remote memory to application space (also multiply by *beta*)
11. ***endfor***

Algorithm: DMAPipeline($C_{i,j}$)
$C_{i,j,k}$: *the* $C_{i,j}$ *blocks are further partitioned into sub-blocks*
//
//mult1
1. calculate $C_{i,j,1}$ in local memory
2. ***for*** each sub-block $C_{i,j,k}$ do //k=2,3…
//store1
3. DMA transfer $C_{i,j,k-1}$ from local memory into remote memory
//mult1
4. calculate $C_{i,j,k}$ in local memory
5. ***endfor***
//store1
6. DMA transfer the last $C_{i,j,k}$ from local memory into remote memory

算法4. 结合数据存储优化（Data Placement）的 DGEMM

因此我们把矩阵 ***C*** 直接输出到远程存储的操作从 *mult* 核心程序中分离出来。核心程序产生的结果输出到 GPU 本地存储，而非远程存储。如算法 4 所示，先前的 *mult* 操作分成两个阶段：$mult_1$ 和 $store_1$，$store_1$ 将 ***C*** 矩阵从 GPU 本地存储传输到远程存储。资源的占用情况发生改变，$mult_1$ 完全由 GPU 设备执行，而 $store_1$ 由 DMA 引擎传输数据。由于核心程序和 DMA 操作都是异步执行，我们在显存中同样采用双缓存策略来存储 ***C*** 子块，使得这两个操作并行执行。为了表述更清晰，以下部分我们用 $store_2$ 来代替 *store* 操作。

到目前为止，优化的 DGEMM 中建立了 5 级流水线（$load_1$、$load_2$、$mult_1$、$store_1$、$store_2$），各个操作的资源占用情况如表 2 所示。至此我们成功地解决了第 2 节提到的 ACML-GPU 中的两个问题。$mult_1$ 核心程序只需要 GPU 资源，不再需要 PCIe 总线和系统内存，消除了 *mult* 中的 PCIe 竞争，$mult_1$ 核心程序可以与 $load_2$ 并行执行。算法 4 不仅提出了更快速的核心程序执行，而且细化了流水线，降低了由于资源竞争造成的流水线延迟。我们绘出了流水线的粗略时空图（图 4），此图只画出了流水线的大致流程，未考虑每个微小步骤之间的资源冲突情况。我们用不同的底纹表示这五个操作。$mult_1$ 块中的条形小块表示被 $mult_1$ 核心程序重叠的子矩阵的数据传输操作。如此图所示，除了流水线的开始和结束开销，算法 4 中大部分数据传输过程被完全重叠。

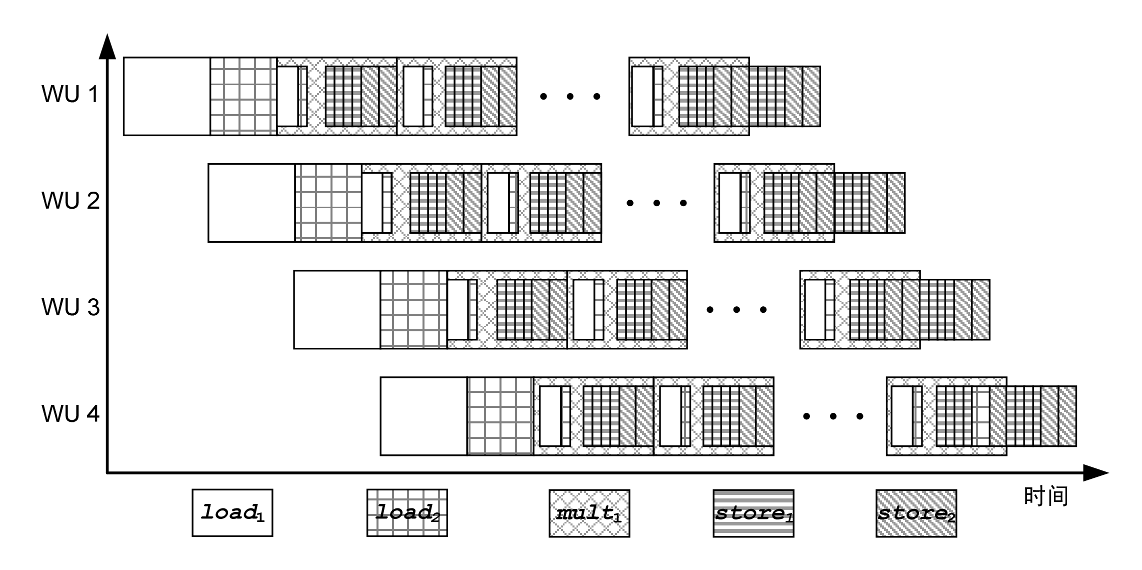


图4. 优化后的 DGEMM 流水线执行时空图

4 实验结果和分析

4.1 实验建立

我们的实验平台是 2-路 Intel Xeon 5650 CPU 和一个 ATI HD5970 GPU 卡组成的异构平台。表 3 概括了实验平台的配置参数，其中英特尔的多核 CPU 双精度浮点峰值性能为 128GFLOP/s。CPU 内存大小为 24GB，带宽为 31GB/s。GPU 集成了两个 Cypress 芯片，双精度浮点峰值性能为 928GFLOP/s。GPU 显存大小为 2GB，带宽为 256GB/s。该异构系统双精度浮点峰值性能达到 1056G FLOP/s。

表3. 实验平台的系统配置参数

处理器		Intel Xeon X5650	ATI HD5970
架构		Westmere EP	Cypress
频率		2.66Ghz	725Mhz
芯片数		2	2
双精度浮点峰值		128GFLOP/s	928GFLOP/s
DRAM	类型	DDR3 1.3Ghz	GDDR5 1.0Ghz
	大小	24GB	2GB
	峰值带宽	31.2GB/s	256GB/s
PCIe2.0		x16, 8GB/s	
编程环境		icc + openmpi	ATI Stream SDK 2.2

为了给出完整的实验分析，我们依次运行了第 3 节的三个优化算法，其中每个优化算法都包含前一种优化。为方便描述，我们定义一些记号代表不同的优化版本。

- DB：该程序执行算法 2——利用双缓存策略隐藏从远程存储写回到 ***C*** 矩阵到应用空间的开销。程序分配的两个缓存大小由显存中的矩阵大小决定。
- DR：该程序执行流水线算法 3，基于 DB 优化并对降低输入矩阵的载入开销进行了改进，其中一个重要优化是开发读入矩阵的数据重用。
- DP：在 DB 和 DR 的基础上，该程序执行算法 4，针对 CAL 系统存储层次中的数据存放进行优化。其主要优化是改变了矩阵 ***C*** 的存储位置并利用 DMA 方式设计出更有效的流水线。
- HB：上述三个程序单纯利用 GPU 的计算资源，本程序实现了混合版

DGEMM——CPU 和 GPU 并行进行矩阵乘法运算。我们采用引文[21]中提出的 CPU 与 GPU 之间负载均衡策略，在 CPU 和 GPU 之间进行矩阵划分。实验中，我们启动两个 MPI[7]进程，每个进程使用一对 CPU/GPU。

这四个程序代表了四种优化策略，程序之间的优化策略包含情况为 DB<DR<DP<HB。由于 ACML-GPU[7]库中的 DGEMM 代码是开源的，我们利用这个代码作为我们的初始实验和实验对比来衡量本文的优化效果。

表 4 给出了实验中用到的矩阵大小(***m***, ***n***, ***k***)。由于 ***m***、***n*** 值的不同对 DGEMM 性能几乎没有影响，因此我们设定 ***m***=***n***。***k*** 的大小决定了输入矩阵 ***A***、***B*** 时的数据重用个数，影响 DGEMM 性能。我们取三种 ***k*** 的值代表不同的数据集大小，***k*** 分别等于 1536、2048、4096。在以下部分，我们通过计算相同 ***k*** 值不同 ***m***(或 ***n***)规模矩阵的性能均值得出不同 ***k*** 大小对应的三种规模 DGEMM 性能值。对于详细的剖析我们默认以 ***k***=2048 为例，不同的矩阵规模通过 x 轴上的 ***m***(***n***)值表示。

表4. 实验中使用的 m, n, k 大小

<table>
<tr><th>k</th><th colspan="5">m=n</th></tr>
<tr><td>1536</td><td rowspan="3">16384</td><td rowspan="3">20480</td><td rowspan="3">24576</td><td rowspan="3">28672</td><td rowspan="3">32768</td></tr>
<tr><td>2048</td></tr>
<tr><td>4096</td></tr>
</table>

4.2 实验结果

首先，我们给出优化的 DGEMM 在异构系统上的性能。性能对比的基准程序为 ACML-GPU v1.1，图 5 画出了最终优化版本 DP 和 CPU/GPU 协同计算的混合版 DGEMM（HB）的性能提升。混合版 DGEMM（HB-2GPU）最高性能为 844GFLOP/s 在矩阵规模为(***m***, ***n***, ***k***) = (16384, 16384, 4096)处取得，其相应的效率为 80%。优化后 DGEMM 在 GPU 上的（DP-2GPU）最高性能为 758 GFLOP/s 在矩阵规模(***m***, ***n***, ***k***) = (16384, 16384, 4096)处取得，对应效率为 82%。这些结果说明 DP-2GPU 是 ACML-GPU 库中 DGEMM 性能的 2 倍,混合版 DGEMM 性能进一步提升了 10%-20%。大体来说，随着矩阵规模增大这三个程序都表现出性能和效率的提升。有的规模出现了反常的情况（当 ***m*** = ***n*** =10240 时），因为此时的问题规模不是最优的数据传输和核心程序执行时矩阵块大小的整数倍。随着矩阵规模增大，DP 相对 ACML-GPU 的加速比降低。当 ***k*** 为 1536、2048、4096，加速比分别为 2.9、2.1、1.9 倍。这是由于数据传输与核心程序执行时间的比例随着问题规模的增大而降低。而我们的流水线优化是针对降低 CPU 与 GPU 之间数据传输的开销，因此规模大的矩阵优化效果减弱。我们观察到大规模矩阵比小规模矩阵在 GPU 上相对而言更容易获得高效率，因为小规模矩阵数据传输时间的比重大。矩阵规模越小，越难得到好的效率。并且，矩阵规模越大，HB-2GPU 的性能提升越快。这是因为矩阵规模越大，CPU 部分 DGEMM 的性能越好，从而提升了 HB-2GPU 的整体性能。

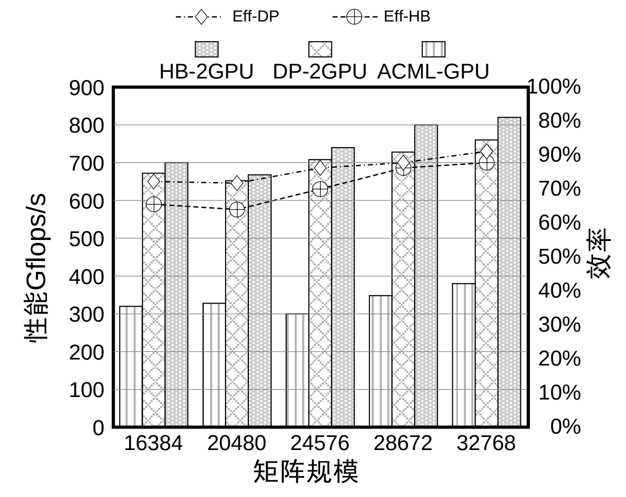


图5. 优化后 DGEMM 性能及效率（两个 MPI 进程）

其次，我们评价了第 3 节提出的三种优化策略。为了屏蔽系统中的其他干扰（如带宽竞

7 Message Passing Interface，消息传递接口

争，我们将在下一节详细介绍），我们在一个 GPU 芯片上运行 DGEMM 程序且不分配计算任务给 CPU，CPU 只用来指导数据传输。图 6 画出了优化后算法性能的增长，包括双缓存优化（DB）、数据重用优化（DR）和数据存储优化（DP）。与算法 1 对比，双缓存优化借助在一个工作单元内部将 $store_2$ 流水执行，性能提升了 16%。数据重用优化由于对不同工作单元间的数据载入操作流水执行，进一步使性能提升了 18%。最后，数据存储优化又更进一步使性能显著提升了 74%，其中我们优化了原始的 *mult* 核心程序并利用 DMA 引擎对 ***C*** 矩阵的写回操作进行流水处理。在单个 Cypress GPU 芯片上，DP 优化算法达到了 408 GFLOP/s 的性能，其效率为 88%。

图 3 中表示的三个数据传输操作（$load_1$、$load_2$ 和 *store*）占据了总计算时间的近 30%，但这并不是全部的数据传输过程，还应加入 $store_1$ 的执行时间。我们优化了流水算法，将 $store_1$ 从 *mult* 核心程序中分离出来。显然，这种做法更符合流水线的本质。将 *mult* 核心程序中的 $store_1$ 计入总数据传输时间后，总数据传输过程占据了总时间的 40%以上。图 7 给出每个数据传输操作占总数据传输时间的百分比。从中看出，图 6 中的每种优化策略得到的性能提升与各部分数据传输时间的分布大体一致，说明我们的优化使流水线得到了充分利用。另外，数据存储优化使得 DGEMM 得到了更多的性能提升，这是因为通过这一优化，核心程序的性能也得以提升。图 6 中还可以看出，优化的 DGEMM 性能对矩阵规模变化不太敏感，性能比较平稳。这一性质为将优化的 DGEMM 扩展到多个 CPU 和多个 GPU 的平台打下了良好基础。本图中在一个 GPU 芯片上 DGEMM 的性能平稳，这与图 5 中异构平台上得到的性能趋势不同。我们将在下一节进一步讨论这一现象。

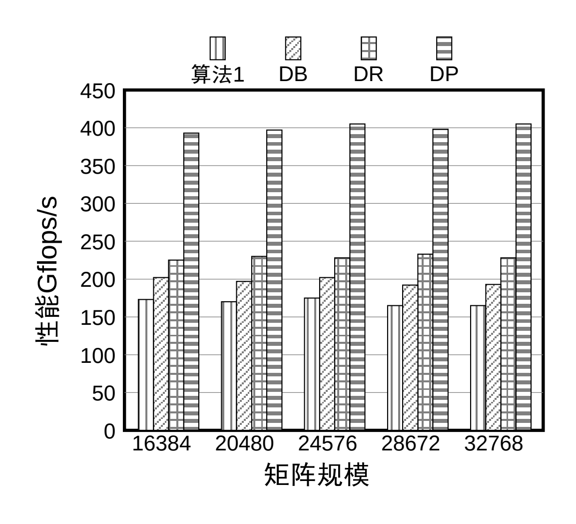


图6. 每种优化方法的性能提升

图7. DGEMM 中四个数据传输操作所占百分比

4.3 实验分析

CPU 数学库中 DGEMM 的效率通常达到 90%以上，异构体系结构上混合版 DGEMM 的效率最高为 82%。本节我们考察：（1）．我们的流水线优化在异构体系结构上还存在多大的优化空间；（2）．体系结构的参数怎样影响优化的 DGEMM 在多个 CPU/GPU 的性能。

4.3.1 性能差距

在流水线执行的五个阶段，$mult_1$ 核心程序决定了 DGEMM 可能达到的最高性能。中里直人[15]优化了 DGEMM 核心程序的性能，在 HD5870（集成一个 Cypress 芯片）上获得的最高效率是 87%。我们采用中里直人优化后的核心程序。不同的是，我们使用了图像读

写寻址模式，而不是全局寻址模式。因此在一个 Cypress 芯片上，我们获得了更高的效率：94%。

图 8 绘出了 DGEMM 与核心程序的效率对比，其中含有一个 GPU 卡优化的 DGEMM 实现（即 DP 实现），两个 GPU 卡的 DP 实现及两个 CPU 与两个 GPU 的混合版 DGEMM 实现。我们对每个测试集求出它们的平均效率。DGEMM 执行过程中多次调用核心程序，我们对每次核心程序计时，最后求出它们的性能平均值作为最终的核心程序性能。优化的核心程序只读写 GPU 的本地存储空间，因此它的性能与 CPU 无关。如图所示，核心程序的平均效率超过 90%（最优效率为 94%），与 CPU 上 DGEMM 效率相近。DP 与核心程序的区别是 CPU 与 GPU 之间通过内存总线和 PCIe 总线的数据传输的有无，本文通过软件流水方法掩盖数据传输的开销。图中的实验结果表示由于数据传输开销，DP-1GPU 的效率相比核心程序降低了 6%。该性能降低有两个原因：首先，流水线的启动和终止时间是不能隐藏的，这部分大约占据了 DGEMM 总时间的 3%。其次，如表 2 所示，优化的 DGEMM 中仍然存在固有的资源冲突，$load_1$ 和 $store_2$ 间的内存总线冲突，$load_2$ 和 $store_1$ 间的 PCIe 总线冲突。除去这两个因素，我们认为优化的 DGEMM 在一个 GPU 芯片上实现 DP-1GPU 几乎达到了最优性能。

从图 8 中还得知，当 DGEMM 运行在更多的 CPU 和 GPU 时（DP-2GPU 和 HB-2GPU）效率下降。当在两个 GPU 芯片上运行 DP 时（DP-2GPU），与 DP-1GPU 相比效率降低 11%。这是因为两个 GPU 芯片上运行的 DGEMM，都需要通过内存总线和 PCIe 总线进行 CPU 和 GPU 之间的数据传输，加剧了这两条总线的资源竞争。另外，当 DGEMM 扩展到两个 CPU 和两个 GPU 的异构系统时（HB-2GPU），效率相比 DP-2GPU 降低 5%。HB-2GPU 中 CPU 运行部分矩阵的 DGEMM，与 GPU 共享同一应用空间，进一步加重了内存总线负担。增加的系统内存竞争使得 HB-2GPU 的效率低于 DP-2GPU。我们将在接下来的两节详细讨论这两种资源竞争。

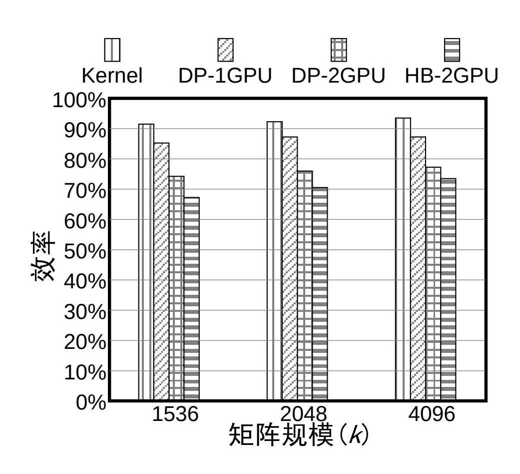


图8. 扩展到多个 GPU/CPU 时优化后 DGEMM 的效率

4.3.2 多个 GPU 上的扩展性

目前大多数加速器（如 GPU, ClearSpeed, Tilera）通过 PCIe 总线与 CPU 相连，主板上的多个 PCIe 插槽可以同时支持多个 GPU 连接。有些 GPU 卡集成了多个 GPU 芯片，如 ATI Radeon HD5970 和 NVIDIA Tesla S1070。因此，优化的 DGEMM 在多个 GPU 上的扩展性问题也很重要。由于实验平台的限制，实验中运行两个 MPI 进程，每个进程负责一个 GPU 芯片的运行。由于影响 DGEMM 扩展性的关键因素是共享资源竞争，如 PCIe 总线和内存总线，用两个 GPU 芯片的性能来预测 DGEMM 在多个 GPU 上的性能是可行的。实验试图通过两个 GPU 芯片之间的带宽竞争来预测多个 GPU 上 DGEMM 的扩展性。实验剖析了 DGEMM 从一个 GPU 芯片到两个 GPU 芯片的有效带宽变化，其中 DP-2GPU 中每个 MPI 进程运行的矩阵规模与 DP-1GPU 相同。从表 2 中可以看出，PCIe 总线和内存总线都存在带宽竞争。

为了突出带宽变化，我们将 DP-2GPU 上的 PCIe 和内存带宽相对 DP-1GPU 的带宽进行归一化。首先，我们考虑 $load_2$ 和 $store_1$ 之间的 PCIe 带宽竞争情况。图 9 显示了平均带宽的降低，y 轴表示归一化后的相对带宽。如图所示，$load_2$ 和 $store_1$ 的有效带宽分别达到 DP-1GPU 的 89%和 56%。由于 PCIe 传输得更加频繁并且需要传输的数据规模更大（矩阵 ***C*** 的大小大于矩阵 ***A*** 与 ***B*** 的大小之和），故 $store_1$ 的带宽下降幅度更大。正如 3.3 节提到的，为充分利用流水线，$mult_1$ 核心程序将子矩阵进一步划分为更小的子块。每个 $store_1$ 操作一个更小的矩阵子块。$mult_1$ 运行时，几个 $store_1$ 同时执行（根据核心程序计算的子块大小，我们的实验中是 4 个 $store_1$ 与 1 个 $mult_1$ 同时执行）。$mult_1$ 执行时间内，PCIe 带宽可认为被 $store_1$ 占用，这样即使在一个 GPU 芯片运行的 DGEMM，$store_1$ 对 PCIe 带宽占用率已经很高。因此，当扩展到两个 GPU 芯片时，PCIe 总线的竞争变得更加激烈。然而，$load_2$ 过程中 PCIe 有效带宽的下降并没有像 $store_1$ 这样严重。这是因为 $load_2$ 与 $mult_1$ 核心程序是工作单元间的流水，而非 $store_1$ 与 $mult_1$ 的工作单元内部流水，因此对 PCIe 的请求不如 $store_1$ 频繁。另外，$load_2$ 传输的矩阵规模为（$\boldsymbol{m}+\boldsymbol{n}$）×$\boldsymbol{k}$，而 $store_1$ 传输的矩阵规模为 $\boldsymbol{m}\times\boldsymbol{n}$。由于 $\boldsymbol{k}$ 比 $\boldsymbol{n}$ 小很多，因此前者对 PCIe 总线的压力较小。

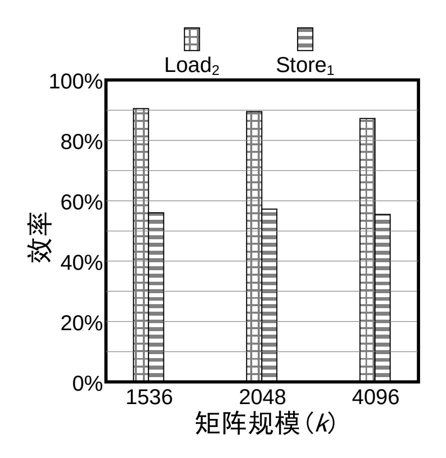


图9. DGEMM 在一个 GPU 卡和两个 GPU 卡上 PCIe 相对带宽百分比

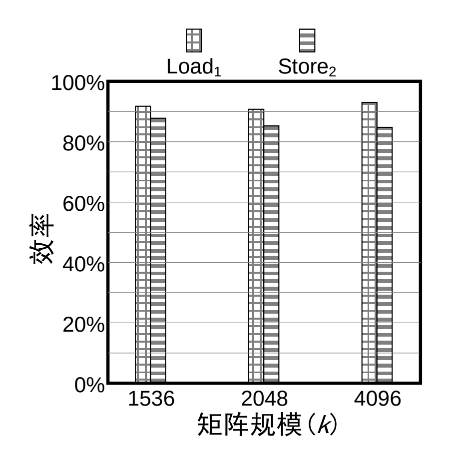


图10. DGEMM 系统内存在一个和两个 GPU 卡上的利用率相对百分比

其次，由于传输数据也发生在应用空间和 CAL 的远程存储上，因此除了 PCIe 带宽竞争，两个 GPU 芯片上还存在内存总线竞争。图 10 表示 $load_1$ 和 $store_2$ 的有效内存带宽分别下降了 8%和 14%。与 PCIe 带宽下降的原因类似，$store_2$ 的带宽下降更明显。

通过对流水线执行过程的分析，我们发现在 DP-1GPU 中，$load_1$、$load_2$、$store_1$、$store_2$ 基本被 $mult_1$ 全部掩盖。然而，图 8 显示有效带宽的降低仍然导致了 DP-2GPU 的效率降低 11%。这说明共享资源的竞争（PCIe 带宽和内存带宽）阻止了部分数据传输过程与 $mult_1$ 核心程序重叠。随着 GPU 数目增加，带宽的请求会更加频繁，加剧带宽竞争对整体性能的影响。综上，当 DGEMM 从 1 个 GPU 芯片扩展到 2 个 GPU 芯片时，PCIe 和系统内存的带宽都下降。基于这些实验结果，我们得到以下两个结论：

- **结论 1：由于 PCIe 带宽的限制，DGEMM 在同一主板上多个 GPU 卡的扩展性受限。** DGEMM 中 $load_2$ 和 $store_1$ 都使用 PCIe 带宽，如图 7 所示，这两个操作占 DGEMM 总数据传输时间的 60%。图 8 的实验结果给出了竞争导致的两个 GPU 芯片上 DGEMM（DP-2GPU）的效率降低。随着 GPU 数目的增多，PCIe 竞争增强，DGEMM 的效率受到的影响增大。

- **结论 2：改善系统内存带宽，GPU-only DGEMM（DP-1GPU 和 DP-2GPU）的性能会有所提升。** 虽然 $load_1$ 和 $store_2$ 都占用内存带宽，但从图 10 得出，它们对内存带宽

的竞争敏感度较低。如图 7 所示，这两部分的执行时间并不是数据传输时间的主要部分。虽然在某些情况下，固定内存的使用可避免 $load_1$ 和 $store_2$ 的竞争。但使用固定内存的前提是数据不会被重新分配，且数据规模小于固定内存的限制。我们的工作证明了这部分的传输开销同样可以通过算法优化来降低。

4.3.3 混合 CPUs 和 GPUs 的扩展性

我们的混合实验平台上，Intel Xeon CPU 提供了 128GFLOP/s 的计算能力，占系统双精度浮点性能的 12%。对计算密集型程序（如 DGEMM）作性能优化时，CPU 的计算能力不容忽视。在混合版 DGEMM 的 HB-2GPU 实现中，矩阵首先被划分为均等的两部分，每部分都分别由一对 CPU/GPU 计算。每对 CPU/GPU 中，我们采用引文[21]中描述的划分算法在 CPU 和 GPU 之间划分任务。虽然 HB-2GPU 相对 DP-2GPU 性能提升了 6%，效率却下降了 5%，本节分析效率下降的原因。

图 8 给出了 DGEMM 在混合 CPUs/GPUs 系统上 DGEMM 的效率。我们在图 11 中分析 CPU 部分（记为 CPU-HB）对 HB-2GPU 的性能贡献。单纯 CPU 版的 DGEMM 程序（记为 Pure-CPU）作为对比程序，Pure-CPU 计算的矩阵规模与 CPU-HB 相同。该图显示混合版 DGEMM CPU 部分性能比 Pure-CPU 降低 22%。这是由于混合版 DGEMM 中 GPU 计算的 DGEMM 也需要从应用空间到 CAL 远程存储拷贝数据，从而对 DGEMM CPU 部分的计算造成干扰，导致 CPU 内存竞争更为激烈。我们从中得出结论：

- **结论 3：改善系统内存带宽有利于降低内存竞争，从而提高混合版 DGEMM 在 CPUs/GPUs 异构系统的性能。**随着 CPU 计算能力的增强，系统内存竞争对混合 DGEMM 的整体性能影响会增大。固定内存的使用将有利于降低内存竞争。

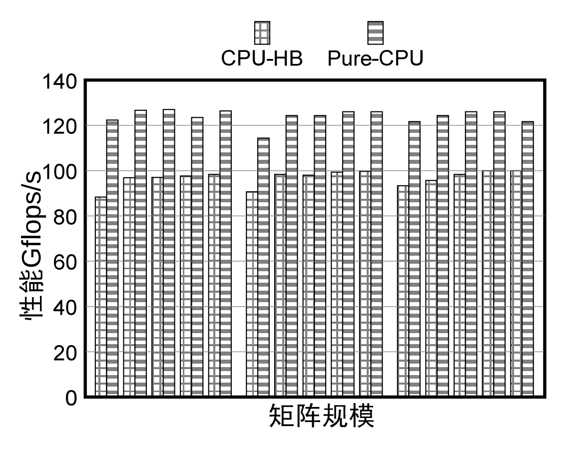


图11. 混合版 DGEMM 中 CPU 部分与纯 CPU 版 DGEMM 性能对比

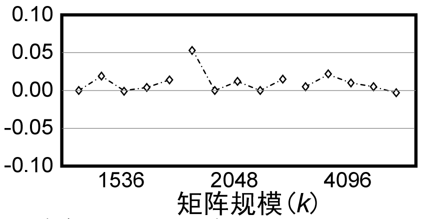


图12. CPU 和 GPU 任务负载不平衡对 DGEMM 的影响

另一个效率降低的原因是 CPU 与 GPU 之间负载的不均衡。我们采用启发式的划分策略，使得 CPU 与 GPU 的执行时间差距在一定阈值内。根据多次实验总结，本文的阈值取为 0.1 秒。图 12 画出 CPU 与 GPU 执行时间的相对差异，纵坐标为（(GPU 计算时间-CPU 计算时间）/CPU 计算时间)，这细微的不均衡性导致了混合版 DGEMM 整体性能的少许降低（约 1%）。

5 其他相关工作

先前有一些工作对矩阵规模小于显存的 DGEMM 进行了 GPU 优化。中里直人基于

HD5870 提出了新的 DGEMM 核心程序[15]，核心程序性能达到了 GPU 峰值性能的 87%。我们优化的 DGEMM 核心程序性能在 HD5970 的一个 Cypress 芯片上达到了 94%。张（音译，Chris Jang）提出的 GATLAS 自动调优器[20]利用自动调优方法加强不同 GPU 结构上的可移植性，并打算在真实应用中调用。GATLAS 同样只关注了矩阵可以存入 GPU 显存的情况，因此现在还没有直接的方式在真实应用中调用 GATLAS。沃尔科夫（V. Volkov）和德梅尔（J. Demmel）在混合 CPU/GPU 系统上实现了单边的矩阵分解（LU, QR 等）[12]，他们将分解过程分配到 CPU 和 GPU 上使其同时执行。但其中的矩阵乘法仍然只针对矩阵能存放在 GPU 显存的情况，不存在 CPU 与 GPU 间的数据传输。此外还有很多工作针对 GPU 上特定程序的性能优化，均未考虑数据传输过程，这里不一一列举。

还有一些工作虽然考虑了数据传输开销，但他们大多集中于实现 CPU 和 GPU 的计算过程并行，并未设法降低数据传输的开销。文卡塔苏布拉玛尼安（S. Venkatasubramanian）和乌杜科（R. Vuduc）在混合 CPU/GPU 架构上实现了雅克比算法[23]。他们考虑了 CPU 与 GPU 之间的数据传输，混合程序的性能提升仅 8%。任达奇（音译， DaQi Ren）和须田玲児（Reiji Suda）在多核 CPU/GPU 系统上实现了大规模矩阵乘法[25]。他们的关注点是能耗问题，利用 CPU 的多线程来进行优化。当一个线程等待 GPU 访存结束的信号时，CPU 开启一个新线程执行其他的任务。这种方法使得 GPU 可以和 CPU 同时进行计算，而数据传输仍然会造成整个执行过程停滞，从而浪费 CPU 和 GPU 的计算能力。费彻廷格（C. Feichtinger）等人在混合 CPU/GPU 集群上实现了并行的格子波尔兹曼（Lattice Boltzmann）方法[24]。他们通过只传输偏微分方程的边界值使传输数据量最小化。他们认为负载不均衡是导致性能提升效果不明显的原因之一。阿加塔（Y. Ogata ）等人提出了基于模型的 CPU/GPU 系统上的异构快速傅里叶变换库[27]。这一模型更好地指导 CPU 与 GPU 之间计算任务的划分。我们在异构系统上优化 DGEMM 时采用引文[21]中的自适应划分算法解决了这个问题。我们的工作关注大规模 DGEMM 在异构 CPU/GPU 体系结构上的实现，其中包含 CPU 与 GPU 之间的数据传输。我们不仅将数据传输计入总时间，而且用流水线算法对其进行优化，使得这部分开销可以与计算重叠。因此，混合版 DGEMM 可以在实际应用中调用。通过我们的优化，混合版 DGEMM 性能达到 844GFLOP/s，对应效率 80%。杨灿群等人提出了用 DGEMM 计算过程掩盖数据传输的开销[21]，采用的优化方法有数据载入流水和数据输出流水。我们还发展了数据存储优化策略改进 DGEMM 核心程序的性能并建立了更细的流水线。这新增的优化策略使 DGEMM 性能提升了 74%，成为最重要的优化手段。另外，我们分析了共享资源（特别是 PCIe 总线和系统内存）的竞争和优化的 DGEMM 在多个 CPU 和多个 GPU 上的可扩展性。通过分析，我们给出了一些对 DGEMM 扩展到异构 CPUs/GPUs 体系结构的建议。

6 结论

我们通过三种策略（双缓存优化，数据重用优化和数据存储优化）优化了大规模 DGEMM，得到了一个新的流水线算法。在流水线中，我们将 DGEMM 核心程序在 GPU 上的执行与数据传输过程重叠。优化的 DGEMM 在一个 ATI HD5970 Cypress 芯片上的性能达到 408GFLOP/s，效率为 88%。在 ATI HD5970 上性能为 758GFLOP/s，效率为 82%。在异构 CPU/ATI GPU 系统上，混合版 DGEMM 性能达到峰值性能的 80% -- 844GFLOP/s。与核心程序的性能对比可以看出，优化的 DGEMM 在一个 GPU 芯片上的效率接近峰值，进一步优化空间不大。然而，当 DGEMM 扩展到多个 GPU 和多个 CPU 时，DGEMM 的效率受到影响，主要影响因素是共享资源的竞争，特别是 PCIe 和系统内存竞争。通过实验和分析，我们得出三个结论：（1）．由于 PCIe 带宽的限制，DGEMM 在同一主板上多个 GPU 卡的扩展性受限；（2）．改善系统内存带宽，GPU-only DGEMM（DP-1GPU 和 DP-2GPU）的性能

会有所提升；（3）．改善系统内存带宽有利于降低系统内存竞争，从而提高混合版 DGEMM 在 CPUs/GPUs 异构系统的性能。

参考文献：

[1] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. *Math. Soft*., 16 (1990), pp. 1--17.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK: A Portable Linear Algebra Library for High-Performance Computers, UT-CS-90-105, May 1990.

[3] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers http://www.netlib.org/benchmark/hpl/

[4] NVIDIA. Compute Unified Device Architecture Programming, Guide Version 3.2,

[5] D. Kirk and W. W. Hwu. ECE 489AL Lectures 8-9: The CUDA Hardware Model, http://courses.ece.illinois.edu/ece498/al/Archive /Spring2007/lectures/lecture8-9-hardware.ppt, 2007.

[6] AMD. ATI Stream SDK CAL Programming Guide v2.0, 2010.

[7] AMD Core Math Library for Graphic Processors (ACML-GPU) http://developer.amd.com/gpu/acmlgpu/pages/default.aspx

[8] NVIDIA. CUDA Community Showcase. http://www.NVIDIA.com/ object/ cuda_apps_flash_new.html.

[9] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley and K. Yelick. Self-Adapting Linear Algebra Algorithms and Software, Proceedings of the IEEE, Volume 93, Number 2, pp 293-312, February, 2005.

[10] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. *ACM Trans.Math. Softw*. 34, 3 (2008), 1–25.

[11] Nath, R., Tomov, S., Dongarra, J. An Improved MAGMA GEMM for Fermi GPUs, University of Tennessee Computer Science Technical Report, UT-CS-10-655 (also LAPACK working note 227), July 29, 2010.

[12] Volkov, V., and Demmel, J. W. Benchmarking GPUs to tune dense linear algebra, 2008 ACM/IEEE Conference on Supercomputing (SC08).

[13] Ryoo, Shane and Rodrigues, Christopher I. and Baghsorkhi, Sara S. and Stone, Sam S. and Kirk, David B. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08), pp. 73-82, 2008.

[14] Ryoo, Shane and Rodrigues, Christopher I. and Stone, Sam S. and Baghsorkhi, Sara S. and Ueng, Sain-Zee. Program optimization space pruning for a multithreaded GPU, Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO'08), pp. 195-204, 2008

[15] N.Nakasato. A Fast GEMM Implementation On a Cypress GPU, 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10). 2010.

[16] H.Wong, M.Papadopoulou, M. Sadooghi-Alvandi, A.Moshovos. Demystifying GPU microarchitecture through microbenchmarking, IEEE International Symposium on Performance Analysis of Systems and Software, March 2010.

[17] G.Tan, Z.Guo, M.Chen, D.Meng. Single-particle 3D Reconstruction from Cryo-Electron Microscopy Images on GPU, 23rd ACM International Conference on Supercomputing (ICS'09), 2009, pp.380-389.

[18] G.Tan, N.Sun and G.R.Gao. Improving Performance of Dynamic Programming via Parallelism and Locality on Multi-core Architectures, IEEE Transactions on Parallel and Distributed Systems, Vol.20, No.2, 2009, pp. 261-274.

[19] Li, Y., Dongarra, J., and Tomov, S. A Note on Auto-tuning GEMM for GPUs. In Proceedings of ICCS'09 (Baton Rouge, LA, USA, 2009).

[20] Jang, C. GATLAS GPU Automatically Tuned Linear Algebra Software, http://golem5.org/gatlas/.

[21] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, Kai Lu, "Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing," cluster, pp.19-28, 2010 IEEE International Conference on Cluster Computing, 2010

[22] J. Dongarra, P. Beckman, Terry Moore, et al. The International Exascale Software Project roadmap. IJHPCA 25(1): 3-60 (2011)

[23] Sundaresan Venkatasubramanian, Richard W. Vuduc Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In Proceedings of the 23rd international conference on Supercomputing (ICS '09). ACM, New York, NY, USA, 244-255.

[24] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, Gerhard Wellein. A Flexible Patch-Based Lattice Boltzmann Parallelization Approach for Heterogeneous GPU-CPU Clusters. CoRR, 2010

[25] DaQi Ren, Reiji Suda, "Power Efficient Large Matrices Multiplication by Load Scheduling on Multi-core and GPU Platform with CUDA," cse, vol. 1, pp.424-429, 2009 *International Conference on Computational Science and Engineering*, 2009

[26] Mark Silberstein, Assaf Schuster, and John D. Owens. Accelerating sum-product computations on hybrid CPU-GPU architectures. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 9. Morgan Kaufmann, August 2011 To appear

[27] Ogata, Y.; Endo, T.; Maruyama, N.; Matsuoka, S.; , "An efficient, model-based CPU-GPU heterogeneous FFT library," Parallel and Distributed Processing, 2008. IPDPS 2008. *IEEE International Symposium* on , vol., no., pp.1-10, 14-18 April 2008

作者简介：

李佳佳： 计算技术研究所 2010 级博士研究生 lijiajia@ict.ac.cn

李兴建： 计算技术研究所 2008 级硕士研究生

谭光明： 计算技术研究所 副研究员

（ 上接第 42 页 ）

[37] C. Bienia, S. Kumar, J.P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications”, in Proceedings of *international conference on Parallel architectures and compilation techniques*, pp:72-81, 2008.

作者简介：

付斌章： 中国科学院计算技术研究所，计算机体系结构国家重点实验室 博士研究生 fubinzhang@ict.ac.cn

韩银和： 中国科学院计算技术研究所，计算机体系结构国家重点实验室 副研究员，硕士生导师

李华伟： 中国科学院计算技术研究所，计算机体系结构国家重点实验室 研究员，博士生导师

李晓维： 中国科学院计算技术研究所，计算机体系结构国家重点实验室 研究员，博士生导师